Optimizing programs with GCC and related tools

Diego Novillo
dnovillo@redhat.com
Red Hat Canada

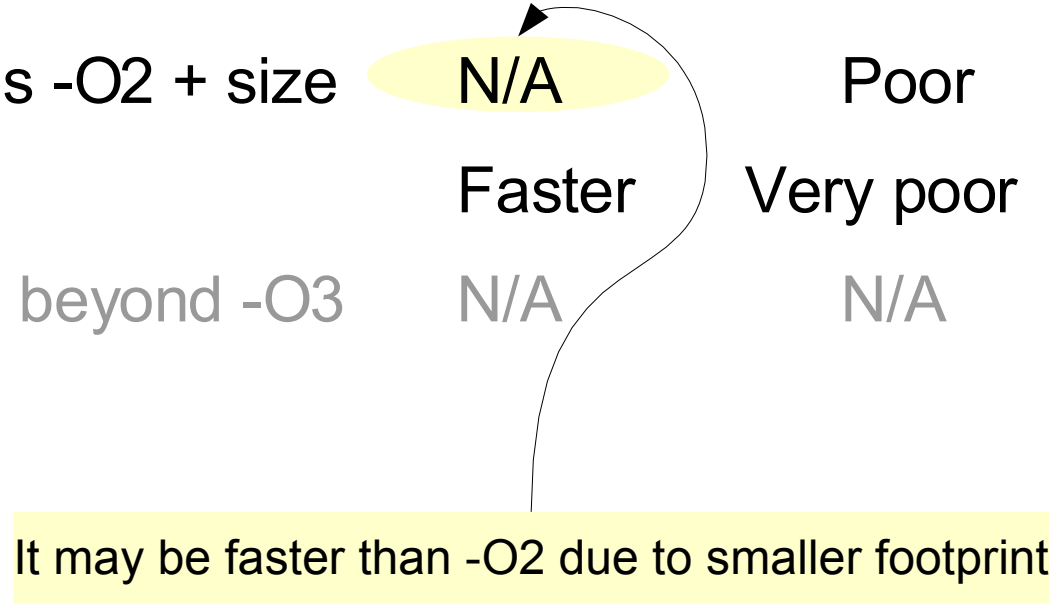
Gelato ICE 2007
San Jose, California
April 2007

Challenges for IA64 optimization

- Itanium platform relies on a compiler for achieving good performance
  - And provides a lot of resources and features for that
- Instruction scheduling, software pipelining, and memory disambiguation are especially important
- GCC has problems in all these areas
  - Instruction scheduling is not aggressive enough
  - Swing modulo scheduling does not work on Itanium
  - Alias analysis on lower IR level (RTL) is weak

Optimization Options

Level	Transformations	Speed	Debuggability
-O0	None (default)	Slow	Very good
-O1	Few	Not so fast	Good
-O2	Many	Fast	Poor
-Os	Same as -O2 + size	N/A	Poor
-O3	Most	Faster	Very poor
-O4	Nothing beyond -O3	N/A	N/A



Optimization Options

- Optimizations done at two levels
  - Target independent, controlled with `-f`
  - Target dependent, controlled with `-m`
- There are more than 100 passes
- Not all can be controlled with `-f/-m`
- `-Ox` is **not** equivalent to a bunch of `-f/-m`
- Use `-fverbose-asm -save-temps` to determine what flags were enabled by `-Ox`
- Use `-fno-...` to disable a specific pass

Enabling additional optimizations

- Not every available optimization is enabled by -Ox

  `-ftree-vectorize`

  `-ftree-loop-linear`

  `-ftree-loop-im`

  `-funswitch-loops (-O3)`

  `-funroll-loops`

  `-finline-functions (-O3)`

  `-ffast-math`

- Hundreds of `-f` and `-m` flags in the documentation

What's good for Itanium?

```
-O3 -ffast-math -funroll-loops -fprefetch-loop-arrays
```

- GCC is still weak in the areas of low-level alias analysis and software pipelining
- What if nothing seems to work?
  - Figure out where time is spent
  - Ask the compiler to show you what it's doing

Profile Guided Optimization

- Three phases
  - Profile code generation: Compile with `-fprofile-generate`
  - Training run: Run code as usual
  - Feedback optimization: Recompile with `-fprofile-use`
- Allows very aggressive optimizations based on accurate cost models
  - Provided that training run is representative!
- Compilation process significantly more expensive
- May not be applicable in all cases

Profiling

- Blindly adding flags will generally not help
- Find out where time is being spent
- Inserting probes: `times`, `getrusage`, `clock`

```
#include <sys/times.h>

compute ()
{
  struct tms t1, t2;
  clock_t ut_elapsed, st_elapsed;

  times (&t1);

  /* Do work */

  times (&t2);
  ut_elapsed = t2.tms_utime - t1.tms_utime;
  st_elapsed = t2.tms_stime - t1.tms_stime;
}
```

GProf

- Probes inserted automatically by compiler
- Compile and link application with `-pg`
- Run application as usual
- Use `gprof` to analyze output file `gmon.out`

```
$ gcc -pg -O2 -o matmul matmul.c
$ ./matmul
$ gprof ./matmul
```

GProf

- Flat profile

```
%       cumulative   self              self     total
 time   seconds     seconds    calls   s/call   s/call name
100.12    443.81   443.81        24    18.49    18.49 matMult
  0.00    443.81     0.00         4     0.00     0.00 timeconversion
  0.00    443.81     0.00         1     0.00     0.00 RunStats
```

- Call graph

```
 index % time    self  children    called     name
                443.81    0.00      24/24     cpuHog [2]
   [1]    100.0  443.81    0.00      24        matMult [1]
   -----------------------------------------------
   [2]    100.0    0.00  443.81                cpuHog [2]
                443.81    0.00      24/24     matMult [1]

  -----------------------------------------------
```

OProfile

- System-wide profiler.
- No modifications to source code
- Samples hardware counters to collect profiling information
- User specifies which hardware counter to sample
- Needs super-user access to start
  - Start Oprofiler daemon
  - Run application
  - Use reporting program to read collected profile

OProfile

```
$ sudo opcontrol --no-vmlinux
$ sudo opcontrol --start
Using default event: CPU_CYCLES:100000:0:1:1
Using 2.6+ OProfile kernel interface.
Running perfmon child on CPU0.
Waiting on CPU0
Perfmon child up on CPU0
...

$ ./matmult
...

$ opreport -l ./matmult
CPU: IA64, speed 1595.68 MHz (estimated)
Counted CPU_CYCLES events (CPU Cycles) with a unit mask of 0x00 (No
unit mask) count 100000
samples  %        symbol name
654198   99.9427  matMult
375       0.0573  anonymous symbol from section .plt
```

OProfile

- When no program is specified, it reports system-wide statistics

```
$ opreport
   CPU: IA64, speed 1595.68 MHz (estimated)
   Counted CPU_CYCLES events (CPU Cycles) with a unit mask of 0x00 (No
   unit mask) count 100000
   CPU_CYCLES:100000|
     samples|      %|
   ------------------
     9331956 90.9284 no-vmlinux
      654573  6.3780 matmult
       93251  0.9086 cc1
       67314  0.6559 libc-2.3.4.so
       23818  0.2321 oprofiled
       23465  0.2286 ld-2.3.4.so
       21638  0.2108 libbfd-2.15.92.0.2.so
       18632  0.1815 bash
```

Using dumps

- Intermediate dumps may be useful to isolate specific transformations
- Three types of dumps
  - High level transformations: `-fdump-tree-all`
  - Low level transformations: `-fdump-rtl-all`
  - Assembly language: `-save-temps -fverbose-asm`

Parallelism in GCC

GCC supports four concurrency models

ILP

- automatic
- no user control
- not intrusive

Vectorization

- automatic
- compiler option
- not intrusive

OpenMP

- manual
- compiler directives
- somewhat intrusive

MPI

- manual
- special libraries
- very intrusive

Ease of use not necessarily related to speedups!

Vectorization

- Perform multiple array computations at once
- Two distinct phases
  - Analysis → high-level
  - Transformation → low-level
- Successful analysis depends on
  - Data dependency analysis
  - Alias analysis
  - Pattern matching
- Suitable only on loop intensive code

Vectorization

- Enable vectorizer

```
$ gcc -ftree-vectorize -O2 prog.c
```

- Additional **`-m`** flags on some architectures
  - PowerPC → **`-maltivec`**
  - x86 → **`-msse2`**
- Speedups depend greatly on
  - Regular, compute-intensive loops
  - Data size and alignment
  - “Simple” code patterns in inner loops
  - Aliasing

Vectorization

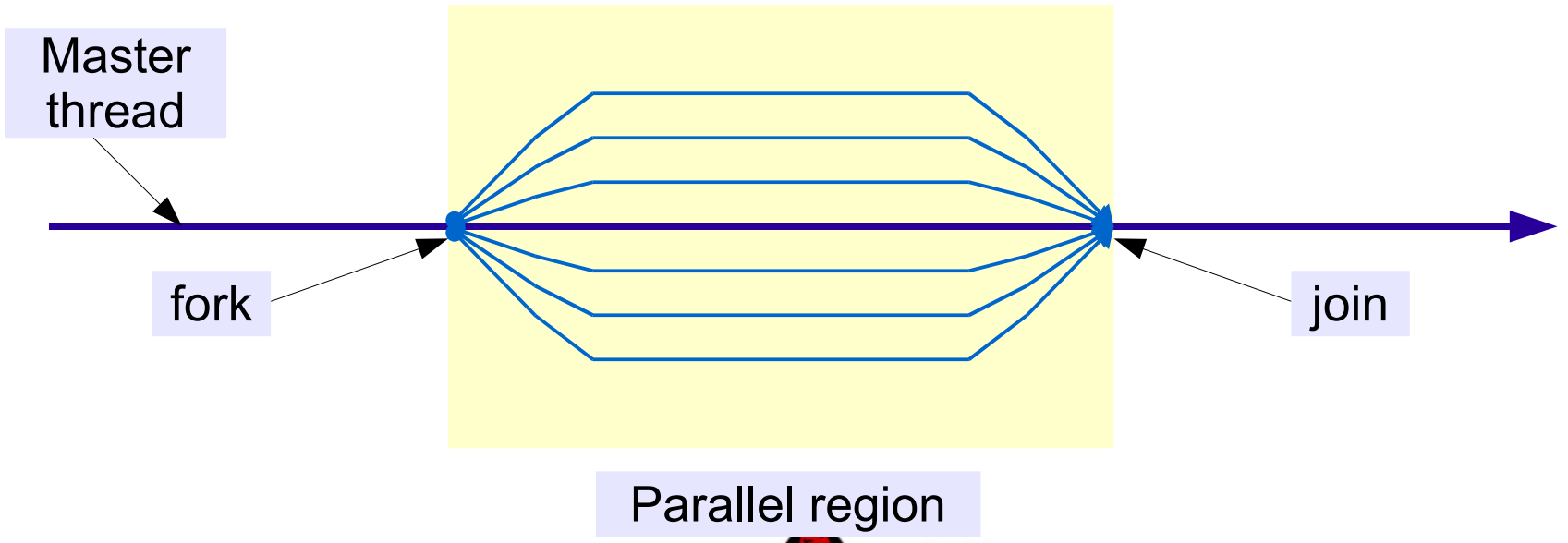
- Debugging
  - `-fdump-tree-vect` enables dump
  - `-ftree-vectorizer-verbose=[0-7]` controls verbosity
- Features and limitations
  - Multi-platform vectorization: x86, ppc, ia64, etc
  - Recognized patterns grow with each release
  - Only works on loops (straight-line code in progress)

OpenMP - Introduction

- Language extensions for shared memory concurrency
- Supports C, C++ and Fortran
- Embedded directives specify
  - Parallelism
  - Data sharing semantics
  - Work sharing semantics
- Standard and increasingly popular

OpenMP – Programming Model

- Based on fork/join semantics
  - Master thread spawns teams of children threads
  - All threads share common memory
- Allows sequential and parallel execution



OpenMP - Programming Model

- Compiler directives via pragmas (C, C++) or comments (Fortran).
- Compiler replaces directives with calls to runtime library (`libgomp`)
- Runtime controls available via library API and environment variables
- Environment variables control parallelism

`OMP_NUM_THREADS` **`OMP_SCHEDULE`**

`OMP_DYNAMIC` **`OMP_NESTED`**

OpenMP - Hello World

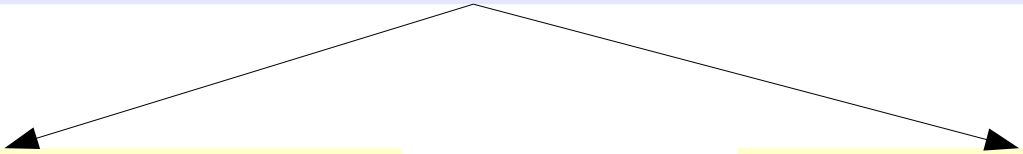
```
#include <omp.h>

main()

{

  #pragma omp parallel

  printf ("[%d] Hello\n", omp_get_thread_num());
}
```



```
$ gcc -fopenmp -o hello hello.c
$ ./hello
[2] Hello
[3] Hello
[0] Hello   ← Master thread
[1] Hello
```

```
$ gcc -o hello hello.c
$ ./hello
[0] Hello
```

OpenMP – Directives and Clauses

- Directives are the main OpenMP construct
- Clauses provide modifiers and attributes to the directives
- General syntax is
  - C/C++

```
#pragma omp directive [ clause [ clause ] ... ]
```

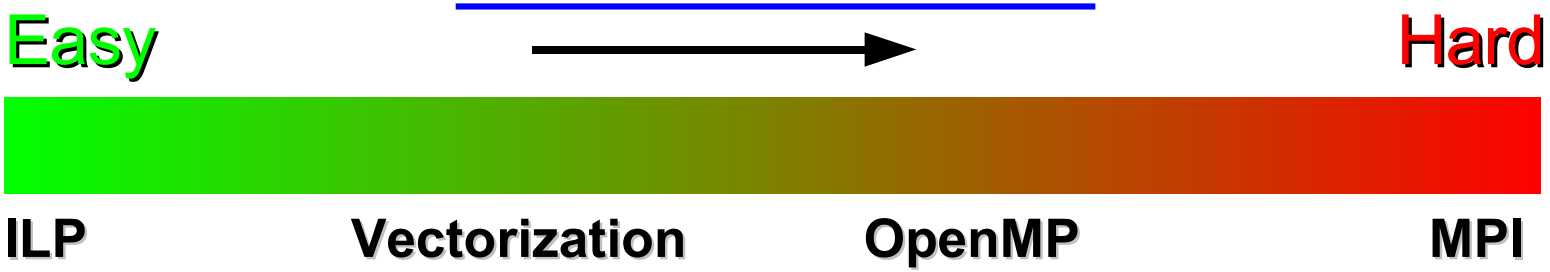
  - Fortran

```
c$omp directive [ clause [ clause ] ... ]
!$omp directive [ clause [ clause ] ... ]
*$omp directive [ clause [ clause ] ... ]
```

OpenMP – Directives and Clauses

- Directives are enabled with **`-fopenmp`**
- Most directives only apply to structured blocks
  - No early exits except program termination
- Directives control
  - Thread creation
  - Work sharing
  - Synchronization
- Clauses control data sharing

Conclusions



- There is no “right” choice
  - Granularity of work main indicator
  - Evaluate complexity ↔ speedup trade-offs
- Combined approach for complex applications
- Algorithms matter!
- Good sequential algorithms may make bad parallel ones

Conclusions

- Performance tuning goes beyond random compiler flags
- Profiling tools are important to study behaviour
- Each tool is best suited for a specific usage
  - Try different flags and use /usr/bin/time to measure
  - Oprofile → system wide
  - Gprof → intrusive but useful to isolate profiling scope
  - Compiler dumps to determine source of problem
- Open source tools for analysis are scarce but growing